

Vorlesung Internetprogrammierung

Message-Driven Beans

Peter Thiemann

Universität Freiburg

Vorlesung Internetprogrammierung, SS2006

- 1 Java Message Service
- 2 Beispiel
- 3 Arten von Nachrichten
- 4 Eine JMS Anwendung
- 5 JMS Hintergrund
- 6 Message-Driven Beans
- 7 Deployment Optionen

Message-Driven Beans (MDB)

Zweck

- Verarbeitung von asynchronen Nachrichten
- Im Gegensatz zum synchronen RMI
- Sender unabhängig von Zustand/Verfügbarkeit des Empfängers

Beispiele

- Webfrontend eines Shops sendet Aufträge als asynchrone Nachrichten an die Auftragsabwicklung (Lager, etc)
- Formatkonvertierung von Webseiten

Nachrichtenquellen

JMS garantiert, andere möglich

JMS — Java Message Service

- Herstellerunabhängige API `javax.jms`
- Zweck: Nachrichtenaustausch zwischen Softwarekomponenten über ein Netzwerk
- Oft auch: Queuing Systeme
- Rollen
 - JMS clients: Anwendungen, die JMS verwenden
 - *producer* oder
 - *consumer*
 - JMS provider: Implementierung von JMS, System zur Zustellung und Weiterleitung von Nachrichten
- Jede Art von Bean kann Nachrichten versenden
- Message-Driven Beans können Nachrichten empfangen

Beispiel: Nachrichten versenden

- Existierendes `TravelAgentEJB` soll Information über ausgestelltes Ticket als Nachricht (an andere Anwendung) verschicken

- Geschäftsmethode `bookPassage` liefert als Ergebnis ein value object `TicketDO`:

```
TicketDO ticket =  
    new TicketDO (customer, cruise, cabin, price);
```

- Zunächst Umwandlung in einen String

```
String ticketDescription = ticket.toString();
```

- Dieser String soll als `TextMessage` verschickt werden

Verbindung zum JMS-Provider

- Zugriff über JNDI
- Generator für Verbindung zum JMS-Provider

```
TopicConnectionFactory factory =  
    (TopicConnectionFactory)  
    jndiContext.lookup  
    ("java:comp/env/jms/TopicFactory");
```

- Verbindung zum JMS-Provider

```
TopicConnection connect =  
    factory.createTopicConnection();
```

- innerhalb einer Verbindung mehrere Sessions möglich
- jede Session ist single-threaded

- Erzeugen einer Sitzung

```
TopicSession session =  
    connect.createTopicSession(true, 0);
```

- Parameter von `createTopicSession`

- `boolean transacted`
- `int acknowledgeMode`

Werte `true` und `0` empfohlen, müssen laut Spezifikation ignoriert werden

- zum Senden/Empfangen werden noch `TopicPublisher` bzw. `TopicSubscriber` benötigt

Zieladresse der Nachricht

- Zieladresse der Nachricht

```
Topic topic = (Topic)
    jndiContext.lookup
    ("java:comp/env/jms/TicketTopic");
```

Nachrichten werden **nicht** direkt an Anwendung verschickt, sondern an

- Topics (vergleichbar zu einer Newsgroup, auch verwendbar für 1-1 Kommunikation) oder
 - Queues
- Zum Senden muss ein TopicPublisher zu diesem Topic erzeugt werden

```
TopicPublisher publisher =
    session.createPublisher(topic);
```


Senden der Nachricht

- Verschiedene Arten von Nachrichten, hier: `TextMessage` mit dem `String` aus der Anwendung als Inhalt

```
TextMessage textMsg =  
    session.createTextMessage();  
textMsg.setText(ticketDescription);
```

- Versenden über den `TopicPublisher`

```
publisher.publish(textMsg);
```

- Verbindung beenden

```
connect.close();
```

Arten von Nachrichten

- JMS Message ist Java-Objekt bestehend aus Kopf (*header*) und Rumpf (*body*)
 - Header: Adressinformation und Metadaten
 - Body: Anwendungsdaten, je nach Art der
- Verschiedene Arten von Nachrichten (Subinterfaces von `javax.jms.Message`)

Typ	Inhalt
<code>TextMessage</code>	String
<code>MapMessage</code>	Namen/Werte-Paare
<code>ObjectMessage</code>	serialisierbares Objekt
<code>BytesMessage</code>	Bytestream
<code>StreamMessage</code>	Stream mit primitiven Werten

MapMessage und ObjectMessage

```
MapMessage mapMsg = session.createMapMessage();
mapMsg.setInt("CustomerID", ticket.customerID);
mapMsg.setInt("CruiseID", ticket.cruiseID);
mapMsg.setInt("CabinID", ticket.cabinID);
mapMsg.setDouble("Price", ticket.price);
```

```
ObjectMessage objMsg =
    session.createObjectMessage();
objMsg.setObject(ticket);
```

JMS-Ressourcen im Deployment Descriptor

```
<session><ejb-name>TravelAgentEJB</ejb-name> ...
  <resource-ref>
    <res-ref-name>jms/TopicFactory</res-ref-name>
    <res-type>javax.jms.TopicConnectionFactory</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>

  <message-destination-ref>
    <message-destination-ref-name>jms/TicketTopic</message-destination-ref-name>
    <message-destination-type>javax.jms.Topic</message-destination-type>
    <message-destination-usage>Produces</message-destination-usage>
  </message-destination-ref>
</session>
```

EJB 2.1

JMS-Ressourcen im Deployment Descriptor/2

- `message-destination-usage`
 - `Consumes`
 - `Produces`
 - `ConsumesProduces`
- Deployer stellt die Verbindung zwischen den logischen Namen aus dem Deployment Descriptor und den wirklichen Ressourcen (JMS-Provider und Topic) her
- Alle Arten von EJB können Nachrichten senden **und** empfangen, aber sie müssen das innerhalb einer Geschäftsmethode tun
- Empfehlung: Nur MDB sollen Nachrichten empfangen, andere EJBs blockieren beim Empfang

JMS Anwendung: Verbindungsaufbau, Sitzung

```
public JmsClient_1
    (String factoryName, String topicName) throws Exception
    Context jndiContext = new InitialContext();

    TopicConnectionFactory factory =
        (TopicConnectionFactory)
            jndiContext.lookup(factoryName);
    Topic topic =
        (Topic) jndiContext.lookup(topicName);
    TopicConnection connect =
        factory.createTopicConnection();
    TopicSession session =
        connect.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
    TopicSubscriber subscriber =
        session.createSubscriber(topic);
    subscriber.setMessageListener(this);
    connect.start();
}
```

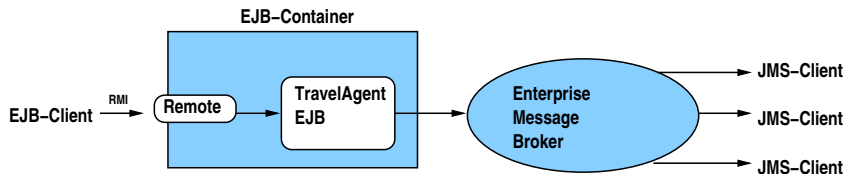
JMS Verbindung: Listener

```
public class JmsClient_1  
implements javax.jms.MessageListener {
```

```
public void onMessage(Message message) {  
    try {  
        TextMessage textMsg =  
            (TextMessage) message;  
        String text = textMsg.getText();  
        System.out.println  
            ("\nRESERVATION:\n" + text);  
    } catch (JMSEException jmsE) {  
        jmsE.printStackTrace();  
    }  
}
```

```
}
```

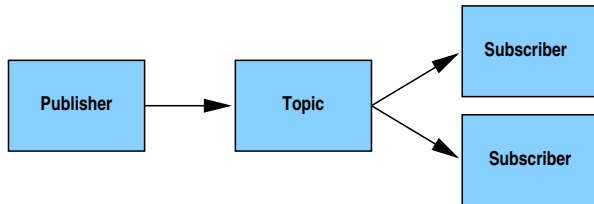
JMS Hintergrund



- Nachrichten sind lose Anbindung
- Keine Übertragung von
 - Sicherheitskontexten
 - Transaktionen
- Transaktion mit JMS-Provider möglich

JMS Verteilungsmodelle

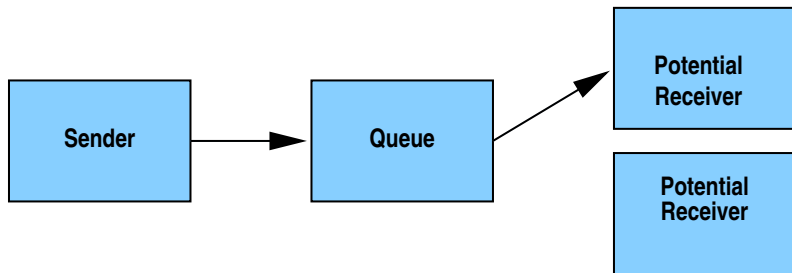
Publish-and-Subscribe



- Ein Sender — viele Empfänger
- Jeder empfängt alle Nachrichten
- Topic ist virtuelle Verbindung
- Empfänger registrieren sich beim Topic (*subscribe*)
- Dauerhafte Registrierung möglich (mit Unterbrechung)

JMS Verteilungsmodelle

Point-to-Point



- Ein Sender
- Viele potentielle Empfänger
- Jede Nachricht gelangt zu einem Empfänger

Programmierung von Queuing

```
QueueConnectionFactory factory =
    (QueueConnectionFactory)
        jndiContext.lookup("java:comp/env/jms/QueueFactory");
QueueConnection connect =
    factory.createQueueConnection();
QueueSession session =
    connect.createQueueSession(true, 0);

Queue queue = (Queue)
    jndiContext.lookup("java:comp/env/jms/TicketQueue");
QueueSender sender =
    session.createSender(queue);
TextMessage textMsg =
    session.createTextMessage();
textMsg.setText(ticketDescription);
sender.send(textMsg);
connect.close();
```

Message-Driven Beans

MDBs

- Server-side Komponenten
- zustandslos
- transaktional

MDB's Container

- Transaktionen
- Sicherheit
- Ressourcen
- Nebenläufigkeit
- Bestätigungen von Nachrichten

Message-Driven Beans/2

Ein MDB besitzt

- EJB Deployment Descriptor
- Bean Klasse implementiert
 - `javax.ejb.MessageDrivenBean`
 - `javax.jms.MessageListener`

Ein MDB besitzt nicht

- EJB Objekt
- Home Interface
- Remote/Local Interface

MDB Interfaces

```
public interface MessageDrivenBean
    extends javax.ejb.EnterpriseBean {
    public void setMessageDrivenContext
        (MessageDrivenContext context)
        throws EJBException;
    public void ejbRemove()
        throws EJBException;
}
```

- MessageDrivenContext ist wie EJBContext
- Manche Methoden werfen Exceptions
 - getEJBHome()
 - getEJBLocalHome()
 - getCallerPrincipal()
 - isCallerInRole()

```
public interface MessageListener {  
    public void onMessage(Message message);  
}
```

- MessageDrivenBean ist nicht fest mit JMS verbunden
- Anderer Nachrichten-Service kann verwendet werden
- Container ruft onMessage () auf
- onMessage () kann Transaktion beinhalten

Antworten auf eine Nachricht

```
public void deliverTicket(Message reservationMsg,
                          TicketDO ticket)
    throws NamingException, JMSException {
    Queue queue = (Queue)
        reservationMsg.getJMSReplyTo();
    QueueConnectionFactory factory =
        (QueueConnectionFactory) jndiContext.lookup
        ("java:comp/env/jms/QueueFactory");
    QueueConnection connect =
        factory.createQueueConnection();
    QueueSession session =
        connect.createQueueSession(true, 0);
    ObjectMessage message =
        session.createObjectMessage();
    message.setObject(ticket);
    session.createSender(queue).send(message);
    connect.close();
}
```


- `activation-config-property:messageSelector`
- Wert ist SQL-WHERE-Ausdruck
- Abfrage von Properties der Nachrichten
- Setzen der Properties durch den Produzent

```
Message message = session.createMapMessage();  
message.setStringProperty  
("MessageFormat", "Version 3.4");
```

Deployment Optionen

Bestätigungsmodus

- Provider erhält Empfangsbestätigung vom JMS-Client
- MDB: durch den Container
- `activation-config-property:acknowledge-mode`
- Werte
 - `Auto-acknowledge`
 - `Dups-ok-acknowledge`

Deployment Optionen

Dauerhafte Registrierung

- Dauerhafte Registrierung hält an, auch wenn die Verbindung zum Clienten temporär unterbrochen ist
- Nachrichten während der Unterbrechung gehen nicht verloren
- `activation-config-property: subscriptionDurability`
- Werte
 - `NonDurable`
 - `Durable`
- Sinnvoll für Topic, nicht für Queue